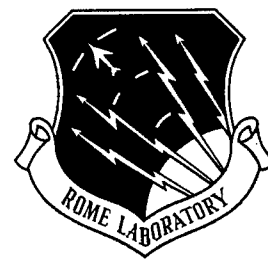


RL-TR-96-79
Final Technical Report
May 1996



A COOPERATIVE PROGRAM UNDERSTANDING ENVIRONMENT

University of Hawaii at Manoa

Alex Quilici and David N. Chin

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

19960730 076

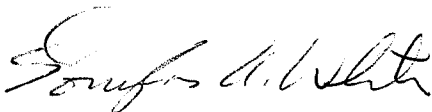
Rome Laboratory
Air Force Materiel Command
Rome, New York

DTIC QUALITY INSPECTED 1

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be releasable to the general public, including foreign nations.

RL-TR- 96-79 has been reviewed and is approved for publication.

APPROVED:



DOUGLAS A. WHITE
Project Engineer

FOR THE COMMANDER:



JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify Rome Laboratory/ (C3CA), Rome NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE May 1996		3. REPORT TYPE AND DATES COVERED Final Sep 93 - Apr 95
4. TITLE AND SUBTITLE A COOPERATIVE PROGRAM UNDERSTANDING ENVIRONMENT			5. FUNDING NUMBERS C - F30602-93-C-0257 PE - 62702F PR - 5581 TA - 27 WU - 76	
6. AUTHOR(S) Alex Quilici and David N. Chin				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) University of Hawaii at Manoa 2540 Dole Street, Holmes 483 Honolulu HI 96822			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/C3CA 525 Brooks Rd Rome NY 13441-4505			10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-96-79	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Douglas A. White/C3CA/(315) 330-2129				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) This report describes research and development undertaken to demonstrate a prototype program understanding environment in which programmers and system cooperate to extract designs from legacy software. The prototype includes several key components: an automated program understanding component; a design notebook for users to extend the automatically extracted design information; and a query answering component. The report describes accomplishments and contributions to the area of program understanding, outlines future work necessary to extend the prototype to be useful in the real applications, and presents some conclusions and lessons learned.				
14. SUBJECT TERMS Software, Understanding, Reverse engineering, Knowledge-based systems			15. NUMBER OF PAGES 40	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Contents

1	Introduction	3
2	Results and Contributions	3
2.1	Automated Program Understanding	4
2.2	Assisted Program Understanding	7
3	Limitations and Future Work	8
3.1	Improving the Automated Program Understander	8
3.2	Addressing Useability Problems with DECODE	9
3.3	Addressing Suitability for Maintenance	11
3.4	Extending DECODE's Notion of Cooperative	13
4	Conclusions	14
A	Plan Definitions	16
A.1	Basic Plan Definitions	16
A.2	Specialized Plan Definitions	18
A.3	Implied Plan Definitions	19
A.4	Some Example Plan Definitions	21
A.5	Constraint Definitions	24
B	Design Editor/Query Browser	28
C	Additional Implementation Details	30
D	Publications Resulting from this Effort	32

1 Introduction

Our original research proposal [Quilici and Chin, 1993] had the following key objective:

Create a demonstration prototype program of a program understanding environment in which programmers and system cooperate to extract designs from legacy software.

This prototype included several key components: an automated program understanding component, a design notebook for users to extend the automatically extract design information, and a query answering component. It also had the following key deliverables:

A prototype of the system and a knowledge base formed from the use of this system to understand a piece of real-world DOD software.

This technical report discusses how well we achieved our key objectives and deliverables. The remainder of the report is organized as follows. Section 2 summarizes what we have accomplished with this project and our specific contributions to the area of program understanding. Section 3 outlines future work that is necessary to extend this prototype towards being useable in the real world. Section 4 presents some conclusions and lessons learned. The report also contains several appendices. Appendix A gives detailed examples of the plans in the plan library and their actual representational format. Appendix B gives a detailed discussion of design editor features and the possible queries users can ask. Appendix C contains additional implementation details. Appendix D enumerates publications that resulted from this effort.

2 Results and Contributions

We successfully created a demonstration prototype of a cooperative understanding environment called DECODE. This prototype is described in detail in [Chin and Quilici, 1995].

The key aspects of our initial stab at cooperative program understanding are:

- The automated program understander (the APU) makes a pass through the source, extracting whatever design information it can.
- The system graphically presents the APU-extracted design information and how it links to the source code. (This is currently done in a pair of windows, one of which shows the design elements, the other of which shows the source code. Highlighting a design element causes the corresponding source to be displayed.)
- The user uses the design editor to extend this design information by adding new design elements, highlighted arbitrary pieces of the source code, and linking them together.
- The user uses the design editor and an additional query component to browse and query the extracted design and its links to the code.

Essentially, our prototype system “cooperates” by providing:

- An initial, partial understanding of the software so that the user need not start from scratch. This is especially important in that the system's recognition of non-contiguous low-level plans provides hints to the user about how non-contiguous source code is related.
- A mechanism for query answering so that the user can not only see how the various parts of the source code are related through the jointly-extracted design, but also which source code is not yet understood and which parts of a design have not yet been located in the source code.

This system makes contributions in two specific areas: algorithms for automated program understanding, and techniques for forming conceptual, machine processable descriptions of existing software.

2.1 Automated Program Understanding

A key contribution of this research is in producing a much improved algorithm for automated program understanding. There is both a straightforward argument for why this algorithm should have better real-world performance than many existing algorithms, as well as some initial empirical evidence that it is a better performer.

Before starting this effort, we had produced an initial program understanding algorithm based on our initial observations of student programmers understanding short C functions [Quilici 1993]. In implementing this algorithm, we based our representation of plans on that of Andersen Consulting's top-down, library-driven Concept Recognizer [Ning et al., 1993], in which plans were represented as a combination of components (which are matched against AST-entries and already recognized plans) and constraints (which describe relationships that must hold between the components). However, we had extended their representation to support a bottom-up, code-driven approach that seemed to better match how the student programmers we initially observed understood programs. In this effort, we further refined the algorithm so that the resulting representation combined three key features: indexing, specialization, and implication.

- *Indexing* involves marking each plan so that the understander considers it only when a particular one of its components is present and one or more of its constraints hold.
- *Specialization* allows a plan to be defined as additional constraints on an existing plan's attributes and considered only when the existing plan is recognized.
- *Implication* allows a plan to be defined as a variant on an existing plan (additional components and/or constraints) and considered only when the existing plan is recognized.

Specific Improvements to the Program Understanding Algorithm

As part of this effort, we addressed several problems with our original algorithm as it first appeared [Quilici, 1994; Quilici, 1993]:

- *The algorithm's handling of indexing was complex.* In particular, the algorithm had to deal with the possibility that at the time it encountered a plan's indexing component in the source code, some subplans that correspond to the plan's other components may not have been recognized. As a result, the algorithm had to maintain partially indexed and partially recognized plans until the missing components were located and the plan could be completely recognized. This led to a relatively complex algorithm with significant space requirements and hard-to-predict performance.

We addressed this problem by organizing the plan library in layers, where each layer consists of plans whose components were plans in previous layers. The algorithm then makes one pass for each layer, looking only for the indexes for that layer. Essentially, this guarantees that the test for whether a plan is actually there can occur at the time a plan is indexed, greatly simplifying the algorithm.

- *There were important relationships between plans we couldn't represent.* In particular, if it was convenient to define one plan as a slight extension (additional component and some constraints) to an existing plan, we couldn't represent that relationship and instead had to represent the variant as a brand-new unrelated plan. This situation occurred because implication was originally unconditional; it was meant to represent the situation where locating one plan automatically meant another plan was present.

We addressed the problem by making implication conditional, where the condition can consist of additional components to look for and additional constraints to test. We also modified implication to take plan library layers into account, so that the recognition of an implied plan takes place when the library layer containing it is processed, and we modified the library so that the conditions that cause a plan to be implied are considered in forming the plan layers.

- *Our definition of plans as specializations of other plans was problematic.* In particular, our original definition of plan specialization allowed us to define a plan as a limited form of conditional implication, in which the conditions all involved trying to check whether various additional constraints held on components or attributes. There are several problems with this approach. One is that this form of specialization is not general enough to represent all key relationships between different plans (such as one plan being just like another except for an additional component). The second is that checking for specialized plans complicated the overall recognition algorithm.

We addressed these problems by generalizing implication and limiting specialization to additional constraints on plan attributes. It turns out that a specialized plan definition can be automatically converted to a regular plan definition, with the plan it specializes as its component and the constraints on that plan's attributes as the constraints on its component. This means the recognition algorithm as a whole need no longer know about specialization, simplifying it significantly. Yet because we generalized implication, we have actually extended the power of our representation.

An Argument for Potential Improved Performance Results

There is an argument for why our program understanding algorithm should perform better than existing algorithms [Chin and Quilici, 1995]. There are three key points:

- *A bottom-up (code-driven) approach should reduce the number of plans considered over a top-down (library-driven) approach.* In particular, a library-driven approach must check for instances of every plan in the library. A code-driven approach checks only for those plans with recognized components. As libraries get larger, individual programs contain a smaller subset of the library, which results in a larger advantage for the bottom-up approach.
- *Indexing should reduce the number of plans considered over the basic bottom-up mechanism.* In particular, in the basic bottom-up approach, the recognition of any plan component leads to an attempt to recognize the entire plan. In the indexed approach, only the recognition of the indexed component(s) leads to an attempt to recognize the entire plan. Thus, indexing should reduce the number of plans considered by the number of times non-indexed components occur.
- *Indexing should reduce the number of constraint evaluations and matches over a non-indexed approach.* In particular, the indexing approach essentially provides a partial ordering of matches and constraint evaluations. An index that severely constrains the possible matches of the remaining components, and that includes constraints with a high failure rate on non-instances of the plan, will reduce the matching over using a randomly selected component as an index.

Interestingly enough, specializations and implication do not directly improve performance; their power is in simplifying the process of providing the plans in the plan library.

Initial Performance Testing

As part of this project, we performed two different tests of our plan recognition algorithm.

One was as part of the DECODE system. We compared our indexed approach against a standard bottom-up approach by using both to recognize various individual plans on a COBOL program with approximately 1000 syntax tree entries. (We recognized individual plans rather than trying to recognize an entire library because resource constraints did not allow to provide a complete plan library suitable for understanding programs of that size.) On average, as reported in [Chin and Quilici, 1995], we achieved the following improvements:

- A ten-fold reduction in the amount of matching.
- A five-fold reduction in the number of constraint evaluations.
- A five-fold reduction (with precomputed dependencies) or two-fold reduction (with dynamically computed dependencies) in the time spent recognizing plans.

The other was a separate implementation that was part of a Master's thesis [Chakravarty, 1995]. This project compared the performance of a particular top-down approach [Kozaczynski et al. 1994; Kozaczynski and Ning, 1992], a bottom-up approach (based on the representation in [Kozaczynski and Ning, 1994]), and the indexing approach on complete recognition of a variety of short C programs with 25-50 syntax tree entries. (The programs were short because we had no C parser or data-flow analyser available. As a result, the student formed the AST and did all the data- and control-flow analysis by hand.) This project

didn't keep meaningful accounting of the amount of matching and constraint evaluation, but instead focused on the total time to perform the necessary recognition. We discovered the following improvements by the indexing algorithm:

- A twenty-fold improvement in the time spent recognizing plans over the top-down approach.
- A ten-fold improvement in the time spent recognizing plans over the bottom-up approach.

2.2 Assisted Program Understanding

This research effort also provides some contributions in the area of assisted program understanding.

One contribution we have made in this area is to provide a mechanism that allows programmers to visually record the design elements they recognize in the code, visually link those elements to the source code that implements them, and then visually browse the source and design in parallel. Our mechanism for recording design information is similar to that of CASE tools. In fact, others have used CASE tools to support reverse engineering by having programmers use these tools to record the design elements they locate within the code. However, existing tools provide no mechanism to link and view the relationships between these design elements and the source. We remedy this drawback by providing users with an explicit mechanism for linking arbitrary blocks of source code to particular design elements, and then allowing users to select a design element to display the source code related to it (and vice versa).

Another contribution we have made is in the area of assisting the user in recognizing design elements. There are two types of assistance.

- *Our automated program understander automatically recognizes some design elements.* This required modifying the plan definition language so that high-level plans are linked to the design elements they implement. That way, when the system recognizes one of these plans, it automatically recognizes the corresponding design element. This provides a connection between the space of plans needed by the automated program understander and the space of design elements recognized by the user. In addition, when the system recognizes a design element, such as an operation, it also recognizes related design elements that must be present, such as the object associated with that operation and the classes to which it is statically related.
- *User can specify the source code related to various design elements by selecting recognized plan instances as well as highlighting source code lines.* Because these plans are sometimes non-local (involving statements dispersed throughout the code), their automatic recognition highlights non-obvious relationships between these statements, and reduces the amount of work necessary for the user to specify the relationships between design elements and source code.

Our final contribution in this area is the mechanism we provide for answering queries about the extracted design and its relationship to the source code. We focus on two general

classes of queries: about the state of the understanding process (such as “What code is not mapped to any design element?” and “What design elements are not mapped to any code?”) and about the relationship of sets of design elements to the code (such as “What code involves this class or any of its subclasses?”). These queries allow the user to see at a glance the underlying relationships between large blocks of source code. Answering these queries is only possible because we record and maintain the links between design elements and the source code. Other systems, such as COBOL-SRE [Ning et al., 1993], which allows users to label highlighted sections of code, cannot support these queries.

3 Limitations and Future Work

3.1 Improving the Automated Program Understander

There are two specific areas where we need to perform further work on our APU: additional empirical tests of its behavior, and formalizing its algorithm in a way that allows detailed comparison of its behavior with other algorithms.

Performing Large Scale Scale-Up Experiments

Our initial performance studies show that our revised program understanding algorithm represents a significant improvement in performance over our implementations of existing algorithms. Unfortunately, because of limitations with our original tools (a lack of parsing or data-flow tools for C, as well as a limited ability to form ASTs and data and control-flow analysis for COBOL), we have done only a few studies, which leaves a crucial open question: Does the apparent five to ten-fold improvement in performance scale? In particular, does it scale with the size of the plan library, with the size of the program, and with the size of individual plans? That is, are we improving the understanding algorithm’s order or just its constant?

To address these questions, we need to perform several tests. One is to take a variety of COBOL programs ranging in size from the 1000 AST element programs with which we’re now working to much larger programs in the 10,000-100,000 AST element range. This is a matter of obtaining the appropriate AST-building and flow-analysis tools (e.g., Software Refinery). This will give us an idea of the scalability of the algorithm in terms of program size. Another test is to build a library containing the estimated 150-200 plans necessary to completely understand a 1000 AST element program. This is mostly a matter of labor; having a student sit down and construct these plans will take on the order of one or two full time months. This will give us an idea of the scalability of the algorithm in terms of the plan library, as well as in terms of plan size (as a realistic library will have some plans with a large number of components).

Formalizing Our Understander’s Behavior

While empirical testing is certainly necessary and is likely to help us better understand our program understanding algorithm’s performance with real-world programs, it doesn’t help us understand theoretically why it performs the way it does, nor does it help us compare its behavior to that of other program understanding algorithms. Part of the problem is

that our algorithm uses heuristic tricks to improve performance (such as indexing), which makes it difficult to analyze its performance or predict how its performance will be affected by variants in the plan library (such as adding large numbers of new plans) or in the programs being understood (such as changing the distribution of basic syntax tree items and the dependency relationships between them). It also uses a representational framework (components and constraints) that differs from frameworks used by some other researchers (such as flowgraphs [Wills, 1992; Wills, 1994] or transformation rules [Johnson, 1986]), which makes it difficult to systematically compare these different algorithms.

One way to partially address the comparability problem would be to find a common framework into which we could transform various program understanding algorithms, including ours. Similarly, we could also try to find a common representation into which we can transform the representations used by these algorithms. This common framework would allow comparison of their behaviors and, if the framework is amenable to analysis, it would also allow us to theoretically analyze and predict their performance.

We have recently begun work in this direction, having transformed our algorithm into a constraint satisfaction framework [Quilici and Woods, 1996]. We have chosen constraint satisfaction because it has the advantage that it has been well studied and that a variety of approaches to solving constraint satisfaction problems have been analyzed theoretically. This initial foray into constraint satisfaction has allowed us to understand better what is happening with our algorithm's approach to indexing in terms of reducing the size of variable domains and in ordering constraints. However, we still need to transform other existing algorithms into this framework so that we can compare their performance, and we need to work on finding a common representational framework. One reasonable approach is to attempt to transform components and constraints into the plan-calculus flow-graph representation used by the Programmer's Apprentice [Rich and Waters, 1990].

3.2 Addressing Useability Problems with DECODE

Our original plan was to test DECODE on an actual DoD COBOL program. However, our initial testing (both with ourselves and with several graduate students) has made clear that DECODE has serious shortcomings that must be addressed before it can be applied to large systems.

Providing Program Plans

For purely automated plan-based program understanding techniques to really pay off on large programs, it is necessary to have both a sizeable domain-independent library (with code patterns for tasks such as maintaining a table) and a significant domain-dependent library (with code-patterns for common tasks such as batch-verification of transactions). Because there is such a large variety of possible domains, it is unrealistic to expect that these libraries can be completely provided in advance. That means that ordinary programmers will need to provide these patterns.

The problem is that DECODE has no provision to allow non-expert users to easily add new code patterns to the library. Providing these plans now requires that the user have specific knowledge of our plan language in terms of its syntax, the specific abstract syntax tree items and plans available as plan components, and the exact semantics and behavior of

a variety of different constraints. In addition, the user must be able to specify the indexing of plans (when they should be considered) and be able to define some plans as slight variants of others (determining which existing plan a new plan is most like). This makes it difficult for an average user to provide program plans.

In particular, it currently takes expert users of the system (the designers), approximately 15-20 minutes to provide a basic plan pattern (e.g., Display-Labelled-Record [Chin and Quilici, 1995]), which consists of 4 components (three MOVEs and a WRITE, in its simplest incarnation) and six constraints. Coming up with the plan requires determining which chunks of code the plan is supposed to match, defining a plan (currently in a LISP-like syntax) that matches these fragments, trying it out, seeing if it recognized more or less code than it should, editing the plan to make it more accurate, and then repeating the process. For more complex plans, such as those requiring constraints between other plans, this process is more difficult because exactly what constraints should hold between higher-level plans with potentially interleaved components is often difficult to determine.

As a result, it is clear that DECODE requires an alternate mechanism for providing plans. Our proposed future mechanism is to let users provide these plans by example. The idea is to have users select a program section that corresponds to an instance of a particular design element (e.g., highlighting a set of statements that together are an instance of the plan Display-Table). This code section can be considered an overly constrained code pattern that will only recognize this one instance. However, the system can provide help to the user in generalizing this code pattern. In particular, the system can:

- Visually display a list of all the specific constraints involved in the example, which the programmer can edit to create a final code pattern.
- Provide information about the frequency of each component's occurrence in the program, as well as about the percentage of various constraints holding among instances of the plan's components, which can help the user select which components and constraints might be the best index.
- Automatically define a new plan in terms of existing plans, when the system has already recognized these plans within sections of code the user has highlighted (as opposed to the underlying syntax tree elements).
- Automatically reconstruct a plan's definition as an implication by examining the user's final plan definition to see whether it contains any existing plan definitions as a subset.
- Allow the user to immediately run the plan against the code and examine its performance. This could be done by letting the user select chunks of code that are expected to match against the plan and then request a match, and by having the system determine exactly where and why the expected match doesn't occur (e.g., which components are missing, which constraints failed, and so on).

Of course, there are many issues involved in making this mechanism work. How can we best display constraints to convey their meaning? What's the best way to let users visually edit constraints and generalize pieces of a plan? How do we decide which plans are most similar to a new plan? There are also more speculative approaches, such as having the user

select a number of examples of a particular plan and having the system attempt to find a suitable generalization that covers those examples.

As a result, this mechanism was not a part of our original proposal, and we have therefore not implemented it in the current version of DECODE. It is, however, a crucial detail in terms of constructing the necessary code pattern library without using specialized knowledge engineers.

Recording Design Relationships

The basic mechanism in DECODE for indicating relationships between source and design is to have the user highlight the relevant lines. From our simple tests, it is apparent that this approach won't scale. Users often fail to highlight crucial portions of the implementation, especially when it is delocalized and consists of plans that are not in the APU's plan library.

This problem can be addressed by providing mechanisms that suggest related code chunks to users. For example, COBOL/SRE [Ning et al., 1993] allowed the users to form *segments* from a variety of different slicing techniques. Users were then allowed to perform various set operations to these segments and then save these segments by name. In some sense, DECODE can be thought of as having a much more primitive mechanism for forming segments (the user simply locates and then highlights the statements in the segment of interest) but a much more advanced mechanism for organizing segments (by allowing users to connect segments to various elements in the design space). We could greatly improve DECODE's scalability by integrating it with COBOL/SRE's segment forming mechanism.

Another complimentary way to address this problem is to integrate the segment forming mechanism with work done on automatically recognizing objects and operations [Newcomb, 1995]. This work is based on slicing and forms a variety of slices that closely correspond to particular classes of methods on objects. For example, a slice through a COBOL procedure that updates one record but uses another can be considered to be a transforming method on the object represented by the updated record. This work suggests mechanisms for locating candidate code for objects and their operations, but does not provide any mechanism for linking these recognized objects to a domain model (that is, it really does not know exactly what type of object or operation it recognized). However, by trying to recognize program plans on candidate "methods", we can greatly limit the search space for the program understanding tool, potentially improving its performance. And by then allowing the user to link these "methods" to actual operations in the design hierarchy, we can greatly simplify the user's task in locating the complete implementation of design elements.

3.3 Addressing Suitability for Maintenance

In addition to useability problems with larger programs, DECODE also has several flaws that currently make it unsuitable as an understanding tool to support real-world program maintenance.

Dealing With Program Modifications

DECODE currently assumes the program being understood will not change—an assumption that clearly does not hold with real-world programs. As a result, DECODE currently has

no mechanism for recognizing which parts of the program (and their corresponding design elements) are affected by a given user change, which makes it unusable as a real-world maintenance tool (although it does not detract from its usefulness as a tool for exploring cooperative program understanding).

Fortunately, this problem is relatively easy to address. We need to modify DECODE so that whenever a program is edited, it performs several specific tasks:

- DECODE must determine which lines were changed. This task essentially maps to doing a “diff” between the original and the changed source.
- It must determine which previously recognized plans are no longer present. This can be done by examining the underlying links between plans and statements and removing any plans that contained deleted or modified lines, as well as their links to design elements.
- It must recognize any new plans that have been added. One obvious approach is to simply rerun the APU on the entire program to form a new understanding and examine the differences between what it recognizes on this pass and what it recognized on previous passes. (We don’t want to simply replace the entire set of recognized plan instances, however, since users may have linked them to design elements.)

In addition, DECODE also needs to somehow make the user aware of what has changed in terms of design elements, such as providing a view of the program’s extracted design that combines multiple versions of the program. This view can then indicate the plans that no longer exist and the design elements to which they were connected, as well as any newly detected plans. This user can then determine whether the modified code is simply an alternative implementation of those design elements or whether the modification reflects a change in design that must be recorded in the design-editor.

Extracting Complete Object-Oriented Designs

DECODE now only extracts and records static design elements: classes, their operations, and the relationships between them. However, this is only a portion of a complete object-oriented design that is needed to reengineer a system. Such a design also includes dynamic models, such as explicit state transition diagrams that show the order in which operations occur, the conditions under which they occur, and so on. These high-level state diagrams represent the program’s control flow at the conceptual level rather than at the statement level.

DECODE needs to be augmented to recognize and record these object-oriented state transition diagrams. Doing so will allow the user to see the relationship between conceptual states and particular statements in the legacy system. For example, the user will be able to obtain answers to questions such as “What happens before (or after) the FilterCopy operation is applied to transaction-file?” This goal involves several key tasks:

- Extending DECODE’s automatic program understanding component to recognize and record conceptual control-flow information (i.e., states and state transitions). One possible approach is to extend DECODE’s APU to use the abstract control- and data-flow

information we maintain about recognized plans to determine the flow relationships between the operations these plans implement, and by extension recognize high-level flow relationships between design elements.

- Augmenting DECODE's design editor to allow users to construct state diagrams, link them to recognized design elements, and to traverse them to see the relationships between these states and the code. This is a relatively straightforward extension to the design editor to capture additional information that current object-oriented case tools allow users to record when constructing a design. The primary difference is, once again, the ability to link design to code.
- Augmenting DECODE's query manager to support queries about the dynamic relationships between design elements. This is also a relatively straightforward extension, in which the primary work is to determine which links must be traversed to address various user queries.

While each of these extensions is significant work, it appears that they fit in well with DECODE's current architecture.

3.4 Extending DECODE's Notion of Cooperative

DECODE's "cooperation" in the design extraction process is now somewhat limited. Its currently cooperates by initially extracting as much design information as it can automatically and by providing an easy to use visual environment for recording and locating extracted design information. One way we can make DECODE more cooperative is by integrating it with some of the segmentation creation techniques discussed earlier. This makes it more cooperative by providing the user with help in deciding what might be a candidate method and in indicating the location of a potential design element in the code.

There are also two other places where we can extend DECODE to be more cooperative. One is by making it more aware of the underlying language itself and using this knowledge to aid the user in specifying portions of design elements. For example, in COBOL, when the user highlights a PERFORM as a part of a recognized design element, DECODE should automatically highlight the PERFORMed statements and include them as part of the design element's implementation. Similarly, when the user extracts the portion of the source code that implements a particular design element, DECODE should also automatically extract the relevant data declarations (this is a variant of slicing, where the user doesn't want the full backwards slice involving a given variable). These are both straightforward extensions given existing tools for extracting data and control flow information.

The other is by having DECODE's recognition process interact with the user. For example, DECODE's understander now essentially recognizes a design element only when it can prove that it is present, and it does this understanding just once, before the user starts examining the program. A reasonable alternative is to have DECODE try to relate user-suggested design elements to the code. For example, if a user indicates that a particular design element is present but doesn't specify where, DECODE can attempt to recognize that design element but with relaxed constraints and by allowing missing components. The idea is that DECODE may not have the correct set of plans in its library to recognize the design element, but variants of those plans might be sufficient to make the connection. This

mechanism would provide a way for the user to test hypotheses that certain elements are present, while providing DECODE with a tractable search space to examine.

4 Conclusions

Our work on this cooperative reverse engineering tool has confirmed several of our initial hypotheses:

- Indexing techniques can apparently be used to significantly improve the performance of program understanding algorithms.
- We can provide users with a tool that allows them to visually record and query design information they have extracted from the source.
- We can integrate automated and assisted program understanding techniques by having a common visual, graph-based representation for any user or system extracted design information.

We have also learned several key lessons about building a cooperative understanding environment that scales.

- Any case-oriented tool for recording design information must be integrated with existing tools, such as those used for program slicing. Otherwise, it is too difficult for users to locate and highlight the code relevant to a particular design element.
- Any automated understanding tool must provide a simple method for providing program plans, otherwise it is too time-consuming for normal users to provide these plans.

These lessons will drive our future research efforts, as we extend DECODE to deal with these issues.

References

- [1] Chakravarty, B. (1995). "A Study in Automated Program Understanding", Master's Thesis, University of Hawaii at Manoa, May 1995.
- [2] Kozaczynski, V. and Ning, J. (1994). "Automated Program Understanding By Concept Recognition." *Automated Software Engineering* 1(1):61-78.
- [3] Kozaczynski, V., Ning, J., and Engberts, A. (1992). "Program Concept Recognition and Transformation." *Transactions on Software Engineering* 18(12):1065-1075.
- [4] Johnson, W. L. (1986). *Intention Based Diagnosis of Novice Programming Errors*. Morgan Kaufman, Los Altos CA.
- [5] Newcomb, P. and Kotik, G. (1995). "Re-engineering Procedural into Object-Oriented Systems", In *Proceedings of the Second Working Conference on Reverse Engineering*. Toronto, Canada, July 1995, pp. 237-249.
- [6] Ning, J., Engberts, A. and Kozaczynski, W. (1993). "Recovering Reusable Components from Legacy Systems by Program Segmentation." In *Proceedings of the First Working Conference on Reverse Engineering*. Baltimore, Md., May 1993, pp. 64-72.
- [7] Quilici, A. and Woods, S. (1996). "Toward A Constraint-Satisfaction Framework for Evaluating Program-Understanding Algorithms", *Journal of Automated Software Engineering*, to appear.
- [8] Chin, D. and Quilici, A. (1996). "DECODE: A Cooperative Program Understanding Environment", *Journal of Software Maintenance* 8(1), to appear.
- [9] Quilici, A. (1994). "A Memory-Based Approach to Recognizing Programming Plans", *Communications of the ACM*, 37(3):84-93.
- [10] Quilici, A. and Chin, D. (1993). "A Cooperative Program Understanding Environment", KBSA Funding Proposal, Rome Labs, NY.
- [11] Quilici, A. (1993). "A hybrid approach to recognizing programming plans." In *Proceedings of the Working Conference on Reverse Engineering*, Baltimore MD, pp. 126-133.
- [12] Rich, C. and Waters R. C. (1990). *The Programmer's Apprentice*. Addison Wesley, Reading MA.
- [13] Wills, L. (1992). *Automated Program Recognition by Graph Parsing*. Ph.D. Thesis, Technical Report 1358, MIT Artificial Intelligence Lab, Cambridge MA.
- [14] Wills, L. (1990). "Automated program recognition: a feasibility demonstration." *Artificial Intelligence* 45(1-2):113-172.

A Plan Definitions

This appendix describes how plans are actually defined to our automated program plan recognizer and provides a set of example plans. This description includes considerable detail beyond what appears in our published papers.

A.1 Basic Plan Definitions

Basic plans are defined using several LISP-macros and in a LISP-like syntax. We use the macro `DEFINE-PLAN` to specify the attributes a particular plan will have, and we use the macro `PLAN-IMPLEMENTATION` to define the plan's actual components and constraints. Figure 1 is a simple example.

```
(DEFINE-PLAN Display-Labelled-Record-Plan (Record Label))

(PLAN-IMPLEMENTATION Display-Labelled-Record-Plan
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES      (Record ?r) (Label ?m))
  (COMMENT          "Print message followed by record")
  (COMPONENTS
    (Provide-Msg      (Move-Event (Source ?m) (Dest ?pr-msg-field)))
    (Provide-Rec      (Move-Event (Source ?r) (Dest ?pr-rec-field)))
    (Clear-Record     (Fill-With-Spaces-Plan (Dest ?pr)))
    (Dump             (Write-Event (Source ?pr)))
  (CONSTRAINTS
    (Msg-Is-Field     (Field ?pr-msg-field ?pr))
    (Dump-Depends-Msg (DataDep Provide-Msg Dump ?pr-msg-field))
    (Label-Is-Constant (Constant ?m))
    (Msg-Depends-Clear (DataDep Clear-Record Provide-Msg ?pr-msg-field))
    (Rec-Is-Field     (Field ?pr-rec-field ?pr))
    (Dump-Depends-Rec (DataDep Provide-Rec Dump ?pr-rec-field))
    (Rec-Depends-Clear (DataDep Clear-Record Provide-Rec ?pr-rec-field)))
  (INDEXES
    (Provide-Msg WHEN Msg-Is-Field Label-Is-Constant Dump-Depends-Msg))
  (IMPLEMENTS
    (DisplayLabelled ON Record)))
```

Figure 1: A basic plan definition.

The `DEFINE-PLAN` specifies that each instance of the plan named `Display-Labelled-Record-Plan` will have two attributes, `Record` and `Label`. The corresponding `DEFINE-IMPLEMENTATION` describes one implementation of that plan and consists of a number of clauses:

- `IMPLEMENTATION` provides a unique label for this implementation. It is used in definitions of other plans that are defined in terms of this plan.
- `ATTRIBUTES` specifies that when an instance of a `Display-Labelled-Record` is recognized and created, the `Record` and `Label` attributes will be assigned the values of the plan variables `?r` and `?m`, respectively.

- **COMMENT** allows the definer of the plan to provide an arbitrary number of strings that describe what the plan does. The user can access this description by clicking on any instance of the plan that appears in the design editor window.
- **COMPONENTS** describes the subplans or syntax tree items that must be recognized to have a potential instance of the plan. It consists of a list of components, where each component consists of the component's name, so that it can be referred to by the plan's constraints, and its body. A component body consists of the label identifying the component (either syntax-tree items, whose names all end in the suffix -Event, or plans, whose names all end in the suffix -Plan), followed by its attributes and their required values (either variables or constants). Variable names shared between component attributes implicitly constrain both attributes to have the same value. Constant values for attributes implicitly constraint the attribute to have the specified value.
- **CONSTRAINTS** provides the specific constraints that must hold between the various components. These constraints and their semantic are enumerated later in this appendix. Each constraint consists of an internal name, which can be used to refer to it in the index for the plan, and the actual description of the constraint, which consists of the constraint's name and a set of arguments. These arguments can be variables, constants, or component names.
- **INDEXES** provides the name of a component whose occurrence in the code signals a possible occurrence of this plan followed by **WHEN** and a list of the names of constraints that must hold for that component to be indexed. A missing **WHEN** means that the component itself is a sufficient index and there is no need for additional indexing constraints.
- **IMPLEMENTS** provides a connection between the plan and design elements. It includes the name of the operation the plan implements and the design element with which that operation is associated. Only plans that can be directly connected to design elements include this clause.

Display-Labelled-Record-Plan recognizes that this code:

```
MOVE SPACES TO PRINT-RECORD
MOVE 'INLAID TRANSACTION' TO PRT-MESSAGE
MOVE TRANSACTION-RECORD TO PRT-REC
WRITE PRINT-RECORD
```

corresponds to this instance:

```
Display-Labelled-Record-Plan
  (Record: TRANSACTION-RECORD, Label: 'INLAID TRANSACTION')
```

The recognition process includes recognizing that the first statement above is an instance of **Fill-With-Spaces-Plan**.

A.2 Specialized Plan Definitions

Specialized plans are defined with the SPECIALIZED-PLAN-IMPLEMENTATION macro, a variant of the PLAN-IMPLEMENTATION macro. It shares the IMPLEMENTATION, ATTRIBUTES, and COMMENT clauses. It also has one new clause, SPECIALIZES, which is used to specify which plan or syntax tree item it specializes. This item is specified with optional values for its attributes, which are implicit specialization constraints. In addition, it also has a CONSTRAINTS clause, which provides additional constraints on the attributes. There are no COMPONENTS or INDEXES clauses.

Figure 2 shows how we define Assign-Constant-Plan as a specialization of a Move-Event, and Fill-With-Spaces-Plan as a specialization of Assign-Constant-Plan.

```
(DEFINE-PLAN Assign-Constant-Plan (Value ?v) (Item ?i))

(SPECIALIZED-PLAN-IMPLEMENTATION Assign-Constant-Plan
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES      (Item ?i) (Value ?v))
  (COMMENT         "Fill a field or record with blanks")
  (SPECIALIZES     (Move-Event (Source ?v) (Dest ?i)))
  (CONSTRAINTS
    (Const-Source (Constant ?v))))

(DEFINE-PLAN Fill-With-Spaces-Plan (Record ?r))

(SPECIALIZED-PLAN-IMPLEMENTATION Fill-With-Spaces-Plan
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES      (Record ?r))
  (COMMENT         "Fill record with blanks")
  (SPECIALIZES     (Assign-Constant-Plan (Value 'Spaces) (Record ?r))))
```

Figure 2: Some specialized plan definitions.

Assign-Constant-Plan recognizes that this code:

```
MOVE ZERO TO RP-INVALID-RECORD
```

corresponds to this instance:

```
Assign-Constant-Plan(Value: ZERO, Item: RP-INVALID-RECORD)
```

Similarly, Fill-With-Spaces-Plan recognizes that this code:

```
MOVE SPACES TO PRINT-RECORD
```

corresponds to this instance:

```
Fill-With-Spaces-Plan(Record: Print-Record)
```

A SPECIALIZED-PLAN-IMPLEMENTATION macro translates into a PLAN-IMPLEMENTATION. This is done by preserving the IMPLEMENTATION, ATTRIBUTES, and COMMENT clauses, by turning the SPECIALIZES clause into a COMPONENTS clause whose name is Sole-Component

and whose value is the provided event or plan, by keeping the CONSTRAINTS clause as is, and generating an INDEXES clause so that the provided component becomes the indexing component and any provided constraints become the indexing constraints.

Figure 3 shows the definitions that result from the specialized plans in Figure 2.

```

(DEFINE-PLAN Assign-Constant-Plan (Value ?v) (Item ?i))

(PPLAN-IMPLEMENTATION Assign-Constant-Plan
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES      (Item ?i) (Value ?v))
  (COMMENT         "Fill a field or record with a constant")
  (COMPONENTS
    (Assign-Constant-Plan-Component (Move-Event (Value ?v) (Item ?i))))
  (CONSTRAINTS
    (Const-Source (Constant ?v)))
  (INDEXES
    (Assign-Constant-Plan-Component WHEN Const-Source)))

(DEFINE-PLAN Fill-With-Spaces-Plan (Record ?r))

(PPLAN-IMPLEMENTATION Fill-With-Spaces-Plan
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES      (Record ?r))
  (COMMENT         "Fill record with blanks")
  (COMPONENTS
    (Fill-With-Spaces-Plan-Component
      (Assign-Constant-Plan (Value 'Spaces) (Record ?r))))
  (INDEXES
    (Fill-With-Spaces-Plan-Component)))

```

Figure 3: The plans into which the specialized plan definitions are translated.

A.3 Implied Plan Definitions

Implied plans are defined with the IMPLIED-PLAN-IMPLEMENTATION macro, another variant of the PLAN-IMPLEMENTATION macro. The definition of an implied plan shares the IMPLEMENTATION, ATTRIBUTES, and COMMENT fields of basic plan implementations. It has an additional IMPLIED-FROM clause which names the plan and implementation from which it is implied. It then has the usual COMPONENTS and CONSTRAINTS clauses, which represent the additional COMPONENTS and CONSTRAINTS that must be present to have an instance of the implied plan.

Figure 4 shows the definition of the plan Act-On-Remembered-Condition-Plan, which can be conditionally implied by Act-On-Condition-Plan.

The Act-On-Condition-Plan simply a specialization of an If-Event which ignores the Else statements. The conditional implication requires the presence of an instance of Remember-Condition-Plan, which is used to detect saving the results of a particular test in a flag variable. This plan recognizes that this code

```

(DEFINE-PLAN Remember-Condition-Plan (Cond Flag Value))

(PLAN-IMPLEMENTATION Remember-Condition-Plan
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES      (Cond ?c) (Flag ?f) (Value ?v))
  (COMMENT          "Set a flag to a value to remember a condition held")
  (COMPONENTS
    (Init-Flag      (Assign-Constant-Plan (Value ?init-v) (Item ?f)))
    (Reset-Flag     (Assign-Constant-Plan (Value ?v) (Item ?f)))
    (Test-Cond      (If-Event (Cond ?c) (Then ?t-seq) (Else ?e))))
  (CONSTRAINTS
    (Tied-Assigns   (DataDep Reset-Flag Init-Flag ?f))
    (Reset-Within-If (ControlDep Reset-Flag Test-Cond ?t-seq)))
  (INDEXES
    (Reset-Flag WHEN Tied-Assigns)))

(DEFINE-PLAN Act-On-Condition-Plan (Cond Actions))

(SPECIALIZED-PLAN-IMPLEMENTATION Act-On-Condition-Plan
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES      (Cond ?c) (Actions ?t-seq))
  (COMMENT          "Do an action if a particular condition is true")
  (SPECIALIZES     (If-Event (Cond ?c) (Then ?t-seq) (Else ?e-seq))))

(DEFINE-PLAN Act-On-Remembered-Condition-Plan (Cond Actions))

(IMPLIED-PLAN-IMPLEMENTATION Act-On-Remembered-Condition-Plan (Cond Actions)
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES      (Cond ?c) (Actions ?t-seq))
  (COMMENT          "Do an action if a remembered condition is true")
  (IMPLIED-FROM     Act-On-Condition-Plan impl-1)
  (COMPONENTS
    (Rememberer     (Remember-Condition-Plan (Cond ?c) (Flag ?f) (Value ?v)))
    (Tester          (Equals-Event (Operand-1 ?f) (Operand-2 ?v))))
  (CONSTRAINTS
    (In-Then        (ControlDep Tester Act-On-Condition-Plan-Component ?c))))

```

Figure 4: An implied plan definition.

```

MOVE ZERO to RP-INVALID-RECORD
...
IF TR-FLIGHT-NUMBER IS NOT NUMERIC
  MOVE '1' to RP-INVALID-RECORD

```

corresponds to this instance:

```

Remember-Condition-Plan
  (Cond: TR-FLIGHT-NUMBER IS NOT NUMERIC, Flag: RP-INVALID-RECORD, Value: 1)

```

The Act-On-Remembered-Condition-Plan would recognize that this code:

```

MOVE ZERO TO RP-INVALID-RECORD
...
IF TR-FLIGHT-NUMBER IS NOT NUMERIC
  MOVE '1' TO RP-INVALID-RECORD
...
IF RP-INVALID-RECORD = '1'
  MOVE SPACES TO PRINT-RECORD
  MOVE 'INLAID TRANSACTION' TO PRT-MESSAGE
  MOVE TRANSACTION-RECORD TO PRT-REC
  WRITE PRINT-RECORD

```

corresponds to the following instance:

```

Act-On-Remembered-Condition-Plan
Cond:    TR-FLIGHT-NUMBER IS NOT NUMERIC
Actions: MOVE SPACES TO PRINT-RECORD
          MOVE 'INLAID TRANSACTION' TO PRT-MESSAGE
          MOVE TRANSACTION-RECORD TO PRT-REC
          WRITE PRINT-RECORD

```

A.4 Some Example Plan Definitions

This section provides a collection of additional example plans and code fragments they recognize.

Figure 5 contains some useful plans defined as specializations of other plans. These include Make-One-Plan and Make-Zero-Plan, which are used to represent assignments of '1' and ZERO respectively.

```

(DEFINE-PLAN Make-One-Plan (Field ?f))

(SPECIALIZED-PLAN-IMPLEMENTATION Make-One-Plan
 (IMPLEMENTATION impl-1)
 (ATTRIBUTES (Field ?f))
 (COMMENT "Assign field the value 1")
 (SPECIALIZES (Assign-Constant-Plan (Value '1') (Record ?f))))

(DEFINE-PLAN Make-Zero-Plan (Field ?f))

(SPECIALIZED-PLAN-IMPLEMENTATION Make-Zero-Plan
 (IMPLEMENTATION impl-1)
 (ATTRIBUTES (Field ?f))
 (COMMENT "Assign field the value 0")
 (SPECIALIZES (Assign-Constant-Plan (Value ZERO) (Record ?f))))

```

Figure 5: Some additional specializations.

Figure 6 contains the plans to recognize several variants of reading a record and noting when the end of file is reached.

Read-Record-Plan is the plan of opening a file, reading a record from it, doing something if the end of file is detected end, and closing the file. This plan recognizes this code:

```

(DEFINE-PLAN Read-Record-Plan (File ?f) (At-End ?actions))

(DEFINE-PLAN Read-Record-Plan
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES      (File ?f) (At-End ?seq))
  (COMMENT         "Read record and do some actions for last record")
  (COMPONENTS
    (Opener        (Open-Event (File ?f)))
    (Reader         (Read-Event (File ?f) (At-End ?seq)))
    (Closer         (Close-Event (File ?f))))
  (CONSTRAINTS
    (Open-First    (DataDep Reader Opener ?f))
    (Close-Last    (DataDep Closer Reader ?f))))

(DEFINE-PLAN Read-Record-EOF-Plan (File Flag Value))

(IMPLIED-PLAN-IMPLEMENTATION Read-Record-EOF-Plan
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES      (File ?file) (Flag ?flag) (EOF-Value ?v))
  (COMMENT         "Read record and remember EOF by setting flag")
  (IMPLIED-FROM    Read-Record-Plan impl-1)
  (COMPONENTS
    (Recorder       (Assign-Constant-Plan (Item ?flag) (EOF-Value ?v))))
  (CONSTRAINTS
    (In-AtEnd       (ControlDep Recorder Read-Record-Plan-Component ?seq))))

(PLAN-IMPLEMENTATION Read-Record-Bail-EOF-Plan
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES      (File ?file) (Flag ?flag) (EOF-Value ?v))
  (COMMENT         "Read record and remember EOF by setting flag and"
                  "leaving the current paragraph")
  (IMPLIED-FROM    Read-Record-Bail-EOF-Plan impl-1)
  (COMPONENTS
    (Bailer         (Go-To-Event (Label ?label))))
  (CONSTRAINTS
    (Goto-In-AtEnd  (ControlDep Bailer Read-Record-Plan-Component ?seq))))

```

Figure 6: The plan Read-Record-Plan.

```

OPEN TRANSACTION-FILE
...
READ TRANSACTION-FILE AT END MOVE '1' TO RP-END-OF-TRANS
...
CLOSE TRANSACTION-FILE

```

is this instance:

```

Read-Record-Plan
  File:      TRANSACTION-FILE
  At-End:    RP-END-OF-TRANS = '1'

```


Read-Record-EOF-Plan is an extension of this plan whose actions involve recording that EOF occurred. It recognizes this instance from the code above:

```
Read-Record-EOF-Plan
  File:      TRANSACTION-FILE
  Flag:      RP-END-OF-TRANS
  EOF-Value: '1'
```

Recognizing this instance also requires recognizing that the above MOVE is instance of an Assign-Constant-Plan. The final plan, Read-Record-Bail-EOF-Plan, recognizes this instance:

```
Read-Record-Bail-EOF-Plan
  File:      TRANSACTION-FILE
  Flag:      RP-END-OF-TRANS
  EOF-Value: '1'
```

from this code:

```
OPEN TRANSACTION-FILE
...
READ TRANSACTION-FILE AT END MOVE '1' TO RP-END-OF-TRANS
GO TO READ-TRANSACTION-EXIT
...
EXIT.
...
CLOSE TRANSACTION-FILE
```

Figure 7 contains several other input-reading plans that deal with reading multiple records. The first, Read-Process-Records-Plan, captures the notion of reading input until some termination condition occurs. The second, Read-Process-All-Records-Plan, is implied from the first, and captures the notion of reading all the input records.

Figure 8 contains two other high-level plans: Validate-Record-Plan, which captures the notion of reading a record, performing a test, and printing a message if the test succeeds, and Validate-Records-Plan, which captures validating a number of input records.

Validate-Record-Plan recognizes that this code:

```
READ TRANSACTION-FILE AT END MOVE '1' TO RP-END-OF-TRANS
...
IF RP-INVALID-RECORD = '1'
  MOVE SPACES TO PRINT-RECORD
  MOVE 'INLAID TRANSACTION' TO PRT-MESSAGE
  MOVE TRANSACTION-RECORD TO PRT-REC
  WRITE PRINT-RECORD
```

corresponds to this instance:

```
Validate-Record-Plan
  Record: TRANSACTION-RECORD,
  Test:   RP-INVALID-RECORD = '1',
  Message: 'INLAID TRANSACTION'
```

```

(DEFINE-PLAN Read-Process-Records (File Actions Cond))

(IMPLIED-PLAN-IMPLEMENTATION Read-Records-Plan
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES (File ?f) (Actions ?acts) (Cond ?cond))
  (COMMENT "Read and process records (potentially until EOF)")
  (IMPLIED-FROM Read-Record-EOF-Plan)
  (COMPONENTS
    (Looper (Loop-Event (Cond ?cond) (Actions ?acts))))
  (CONSTRAINTS
    (Read-In-Loop (ControlDep Reader Looper ?acts))
    (Open-Before (ControlDep Looper Opener))
    (Close-After (ControlDep Closer Looper))))

(DEFINE-PLAN Read-Process-All-Records-Plan (File Actions))

(IMPLIED-PLAN-IMPLEMENTATION Read-All-Records-Plan
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES (File ?file) (Actions ?acts))
  (COMMENT "Read and process all records")
  (IMPLIED-FROM Read-Records-Plan)
  (COMPONENTS
    (Tester (Equals-Event (Op-1 ?flag) (Op-2 ?v))))
  (CONSTRAINTS
    (EOF-In-Tester (ControlDep Tester Looper ?cond))
    (EOF-Test (Equivalent ?cond Tester))))

```

Figure 7: The plan Read-Process-Records.

Validate-Records-Plan recognizes that this code:

```

READ TRANSACTION-FILE AT END MOVE '1' TO RP-END-OF-TRANS
...
IF RP-INVALID-RECORD = '1'
  MOVE SPACES TO PRINT-RECORD
  MOVE 'INLAID TRANSACTION' TO PRT-MESSAGE
  MOVE TRANSACTION-RECORD TO PRT-REC

```

corresponds to this instance:

```

Validate-Records-Plan
File:    TRANSACTION-FILE
Test:    RP-INVALID-RECORD = '1'
Message: 'INLAID TRANSACTION'

```

A.5 Constraint Definitions

Our constraints are not necessarily binary. They are binary only when the arguments to these constraints are fully instantiated. In that case, they return T if the constraint holds and nil if it does not. However, when the arguments to these constraints are uninstantiated

```

(DEFINE-PLAN Validate-Record-Plan (Record Test Message))

(PLAN-IMPLEMENTATION Validate-Record-Plan
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES      (Record ?record) (Test ?cond) (Message ?m))
  (COMMENT         "Read record, perform test, and write an error"
                   "message if the test fails")

  (COMPONENTS
    (Reader        (Read-Record-Plan (File ?file)))
    (Validator      (Act-On-Condition-Plan (Cond ?cond) (Actions ?acts)))
    (Notifier       (Display-Labelled-Record-Plan (Label ?m) (Record ?r))))
  (CONSTRAINTS
    (Cond-Display   (ControlDep Notifier Validator ?acts))
    (After-Read     (ControlFlow Reader Validator)))
  (IMPLEMENTS
    (Validate ON Record)))

(DEFINE-PLAN Validate-Records-Plan (File Test Message))

(IMPLIED-PLAN-IMPLEMENTATION Validate-Records-Plan
  (IMPLEMENTATION impl-1)
  (ATTRIBUTES      (File ?file) (Test ?cond) (Message ?m))
  (COMMENT         "Validate many records on a file")
  (IMPLIED-FROM Validate-Record-Plan impl-1)
  (COMPONENTS
    (Looper         (Loop-Event (Cond ?cond) (Actions ?acts))))
  (CONSTRAINTS
    (Read-In-Loop   (ControlDep Reader Looper ?acts))
    (Test-In-Loop   (ControlDep Validator Looper ?acts))
    (Print-In-Loop  (ControlDep Notifier Looper ?acts))))

```

Figure 8: The plan Validate-Record-Plan.

(such as a variable) or partially instantiated (such as the name of a component containing variables), the constraints return a binding list of values for which the constraint would hold.

Our constraints fall into two general categories: non-flow-oriented and flow-oriented. The four non-flow-oriented constraints are Field, Record, Constant, and Equivalent. The two flow-oriented constraints are DataDep and ControlDep.

Field(X,Y)

Field *determines whether X is a field in the record Y*. If both X and Y have been instantiated, this constraint returns T or nil. If only X is instantiated, Field either returns a binding for Y or nil, depending on whether there is a record containing X as a field. Similarly, if only Y is instantiated, Field either returns a set of possible bindings for X that correspond to the fields in Y, or nil if Y has no fields. If neither X or Y is instantiated, the constraint returns "unevaluable".

This constraint is evaluated by examining a table of fields and records built up from the abstract syntax tree (as part of data-flow analysis) before program understanding begins.

Record(X,Y)

Record determines whether X is a record in the file Y. If both X and Y have been instantiated, this constraint returns T or nil. If only X is instantiated, X either returns a binding for Y or nil, depending on whether there is a file associated with X. Similarly, if only Y is instantiated, Record either returns a set of possible bindings for X that correspond to the records associated with Y, or nil if Y has no records. If neither X or Y is instantiated, the constraint returns "unevaluable".

This constraint is evaluated by examining a table of files and records built up from the abstract syntax tree before program understanding begins.

Constant(X)

Constant determines whether X is a constant value. If X is instantiated, this constraint return T if it is a string literal, a numeric constant, or a variable that is never assigned to after it is given its initial value. If X is not instantiated, the constraint returns "unevaluable".

This constraint is evaluated by inspecting X to see if it is a language constant and by then checking a table of read/write information to determine whether it is potentially modified after it is assigned an initial value. (That table is constructed as part of turning the initial parser-produced abstract syntax tree into an internal representation.)

Equivalent(X, Y)

Equivalent determines whether X and Y are equivalent values. Its primary use is to decide if an item to which a variable has been bound is the identical to a particular component within a plan. It returns T if X and Y can be determined to be equivalent and nil otherwise. If either X or Y is unbound, it returns "unevaluable".

This constraint is evaluated by simple matching of X and Y.

DataDep(X,Y,V)

DataDep determines whether Y is data-dependent on X for value V (that is, whether the value of V used at Y is the same value assigned or used at X). It has different behavior depending on whether X and Y are events or plans, and on whether or not X and Y are fully or partially instantiated.

In the simplest case, when X and Y are fully instantiated with events (that is, X and Y are the names of plan components that have already been matched to specific events in the abstract syntax tree), this constraint simply verifies that V is not changed on any control-path leading from X to Y. It returns T if V is unchanged and nil otherwise.

If X is partially instantiated (that is, X is the name of a plan component that has not been matched to specific events in the abstract syntax tree) and Y is fully instantiated with an event, the current behavior is to return set of bindings for X for which the data-dependency relationship holds on the provided variables. (This is currently computed by

collecting all possible events that match X and then computing the subset of these events on which Y is data-dependent for the provided variable. Should a static dependency database become available, however, the system can instead check the variable's dependencies, a more efficient approach.) The overall behavior is the same if Y is partially instantiated instead of X.

If V is not instantiated (that is, no variable is provided), or if both X and Y are only partially instantiated, DataDep returns "unevaluable". A slightly more complex case occurs if X or Y is instantiated to a plan that has been defined as a direct or indirect specialization of an event. In this case, the data dependency analysis is carried out in terms of the underlying event. (That is, if X is a specialization of a plan that hasn't been instantiated yet, all those plans are rounded up, their specialized events located, and the data dependency analysis computed in terms of those underlying events).

An even more complex case occurs if X involves a fully instantiated plan and Y involves a fully instantiated event. In this case, we recognize a data-dependency on V between Y and X if Y has a data-dependency on V on any underlying event in X that modifies V or, if X never modifies V, on any event in X that uses V. (The system evaluates this constraint by locating all underlying events in X involving V and determining whether Y has a data-dependency on any of these events.) The idea here is that there an event has a data-dependency on a plan for a given variable if the variable's value is computed by one or more steps of the plan. Similarly, if X involves an event and Y involves a plan, then Y is data dependent on X for variable V if every event in Y involving Y is either data dependent on X or on some other event in Y. The idea here is that this means any value for V used by Y is computed by Y or computed by X.

The final case is when X and Y are both plans. In this case, Y is data dependent on X if every event in Y that uses V is data-dependent on X.

When X or Y is a plan, as with events, they can be partially instantiated. In that case, we first find the set of matching plans before checking the data dependency relationships.

None of the definitions for data-dependency relationships between plans consisting of multiple sub-plans or events is entirely satisfactory. On the positive side, however, they tend to give intuitive results to users defining plans, even though a better approach would involve a representation of plans that made all flow relationships explicit rather than our attempt to dynamically compute them.

ControlDep(X,Y) and ControlDep(X,Y,S)

There are two forms of the control dependency constraint.

Its simplest form, ControlDep(X,Y), determines whether Y's execution follows X's execution (i.e., there's no way to execute Y without executing X). Its more complex form, ControlDep(X,Y,S), determines whether Y's execution is included in the sequence of statements S that are a branch of X (i.e., if S is executed, then Y must be executed as well). For example, S could be instantiated to a variable corresponding to the statements in the Then clause of an If-Event or the At-End clause of a Read-Event.

As with data dependencies, there are a variety of possibilities to worry about. The simplest case is that X and Y are instantiated to events. In this case, checking ControlDep(X,Y) corresponds to simply verifying that Y is on any execution path between X and the pro-

gram's exit. This is computed by trying to construct a path (in reverse) through the program's flow graph from the program's exit to X that doesn't include Y. If we can't construct such a path, and we can find a path from Y to X, then Y is control dependent on X. If we're given `ControlDep(X,Y,S)`, then instead of checking from X, we check from the first event in the sequence of statements S associated with X. For all these cases, it returns T if it can demonstrate the specified dependency and nil otherwise.

As before, if X and Y (and, if we're provided an S, the first element of S) are specializations of events, we do these checks on the underlying events.

Another possibility is that X is only partially instantiated to an event and Y is fully instantiated to an event (or vice versa). In this case, the basic constraint is treated as a query for all events of X's underlying event type on which Y is control dependent. (This query is carried out by taking each event with X's underlying type and checking whether Y is control-dependent on that event.) The more complex constraint is treated similarly, except that it locates all events of X's underlying event type and then verifies the control dependency between their S branch. In this case, the constraint returns the list of successful bindings for X.

Still another possibility is that X is a plan and Y is an event. In this case, we consider Y control dependent on X if it is control dependent on any event underlying X. (The idea is that if we have recognized an instance of a plan, we know that its components are present and therefore if the plan is executed, those components will be executed as well. So if Y is control dependent on any of those components, it too will be executed.) Similarly, if X is an event and Y is a plan, then Y is control dependent on X if any event in Y is control dependent on X.

The final case is where both X and Y are plans. In that case, X and Y are considered control dependent if Y is control dependent on any event in X.

When X or Y is a plan, as with events, they can be partially instantiated. In that case, we first find the set of matching plans before checking the control dependency relationships.

B Design Editor/Query Browser

This section describes the functionality provided by our design editor and code browser. The user invokes the system by running `code-browser`. This program initially presents an empty window to the user, with a single horizontal menu bar at the top of the program. This menu contains several entries: File, Edit, Query, Extract, Link, and Other.

The File Menu

The File menu has two entries: Open and Exit.

- Open leads to a dialog box used for identifying the COBOL source file in which the user has an interest. The system opens this file, places the first page in the browser window, and invokes the design editor on the design file associated with this COBOL file (if there isn't one yet, the design editor's initial window is empty). In addition, it invokes the program to construct the AST and the automated program understander, if they have not already been invoked for this file.

Once the window is open the user can begin to highlight lines, which is done by selecting a starting line with the left mouse button, dragging the mouse to the ending line, and then releasing the left mouse button. Selecting the right mouse button unselects a line (or set of lines, if it is dragged over highlighted lines).

- Exit simply causes the code browser and any spawned design editor to terminate.

The Edit Menu

The Edit menu has four entries: Clear Highlight, Clear Text, Highlight All Known, and Reverse Highlight.

- Clear Highlight removes all current highlights. This is useful when the user has selected a variety of discontinuous lines and wishes to start over without anything highlighted.
- Clear Text removes all current highlights directly or indirectly associated with the currently selected element in the design window. This is useful for allowing the user to highlight a portion of an existing object's implementation. The user can select a design element, causing its implementation to be highlighted, and then clear the text associated with various design elements that contributed to that design element.
- Highlight All Known highlights all lines that have been connected to any design element. This is useful for getting a feel for what percentage of the program has been at least partially understood.
- Reverse Highlight highlights the non-highlighted lines and turns off the highlighting for highlighted lines. When combined with Highlight All Known, this provides a quick way to see which parts of a program have not been at least partially understood. Also, when combined with Clear Text, this can be used to determine what parts of the program do not appear to be related to a given design element.

The Query Menu

The Query menu has five entries: Complete Design, Object Class, Operator, Instance, and Implementation.

- Complete Design produces a new scrollable window (called a *report window*) that contains a description of every design element, its relationships to other design elements, and the specific code lines that have been linked to it (directly or indirectly).
- Object Class produces a report window containing the complete description of a highlighted design element of type Class, including all of its operations, and its relationships to other classes. When this item is selected, a dialog window comes up that allows the user to select whether all subclasses of the specified object should be displayed as well.

- **Operation** is similar to **Object Class**, but restricts its description to the operations associated with a given class. As with **Object Class**, when this item is selected, the user can choose whether to display the operations for the subclasses of the relevant object class.
- **Instance** produces a report window showing the all recognized instances of a particular object class. As with the other menu entries, the user can choose whether to see this information for the subclasses.
- **Implementation** produces a report window containing all of the lines implementing the current design element.

Once the user has created a report window, the user is able to save this window into a file.

The Extract Entry

The **Extract** entry is selected after the user constructs a report window. It extracts any highlighted code lines and places them in a separate file (of the user's choosing). The idea is that the user can select a particular design element, which causes all the relevant lines in the source file window to be highlighted, and then use **extract** to place all of the code related to that design element in a single file.

The Link Entry

The **Link** entry is selected after the user highlights a variety of lines in the code browser and then selects a particular design element. When **Link** is selected, the system records a link between the highlighted source lines and the selected design element.

The Other Menu

The **Other** menu has a few choices that will eventually be absorbed into other windows: **Unknown** and **Unconnect**.

- **Unknown** produces a report window showing all design elements that have not yet been connected to the source.
- **Unconnect** disassociates a selected design element from any source that is linked to it.

C Additional Implementation Details

DECODE is currently a collection of several different programs:

- *MeraTalk*, a pre-existing, student-written graphical node and link-based design editor that allows the particular nodes and links to be graphically defined. *MeraTalk* keeps track of all design elements created by the user.

This program is written in C, on top of X and Motif. We made some minor modifications to it to allow it to communicate with other software.

- *CodeBrowser*, a newly-written (as part of this project) source code browser that provides the main graphical interface to the user (the menu entries described in the previous section are produced and managed by this program). It maintains a flat-file textual description of the links between the design elements and the code in a format suitable for MeraTalk to read.

This program is written in C, also on top of X and Motif.

- *CobolParse*, a program to produce AST's from Cobol Programs. This is software written by students in the University of Hawaii's Software Engineering Research Library as part of another project. It is written in C.
- *APU*, a newly-written (as part of this project) program understanding tool. It reads the abstract syntax tree description of a COBOL program, produces a somewhat parser-independent AST, and produces an output file (in the expected format for MeraTalk) showing all design elements and plans it recognized and their relationships to the source code. We have taken the extra step of producing a parser-independent AST, so that in the future we can more easily replace our current parser with a public-domain one.

This program is written in Common Lisp.

D Publications Resulting from this Effort

Journal publications:

- [1] Quilici, A. and Woods, S. (1996), "Toward A Constraint-Satisfaction Framework for Evaluating Program-Understanding Algorithms", *Journal of Automated Software Engineering*, to appear.
- [2] Chin, D. and Quilici, A. (1996). "DECODE: A Cooperative Program Understanding Environment", *Journal of Software Maintenance*, 8(1), to appear.
- [3] Quilici, A. (1994). "A Memory-Based Approach to Recognizing Programming Plans", *Communications of the ACM*, 37(3):84-93.

Conference publications:

- [4] Quilici, A. and Chin, D. (1995). "DECODE: A Cooperative Environment for Reverse-Engineering Legacy Software", in *Proceedings of the Second Working Conference on Reverse Engineering*, IEEE Press, Toronto, CA, pp. 156-165.
- [5] Quilici, A. (1995) "Reverse Engineering of Legacy Systems: A Path Toward Success", in *Proceedings of the 17th International Conference on Software Engineering*, IEEE Press, Seattle, WA, pp. 333-336 (invited position paper).
- [6] Quilici, A. and Chin, D. (1994). "A Cooperative Program Understanding Environment", in *Proceedings of the 9th Annual Knowledge-Based Software Engineering Conference*, IEEE Press, Monterey, CA, pp. 125-132.

Workshop publications:

- [7] Quilici, A. and Woods, S. (1996). "A Constraint-Satisfaction Framework for Evaluating Program-Understanding Algorithms", ICSE-96 Workshop on Program Comprehension, Berlin, Germany.
- [8] Quilici, A. (1995). "Toward Practical Automated Program Understanding", in *The IJCAI-95 Workshop on AI and Software Engineering: Breaking the Toy Mold*, Montreal, CA, pp. 87-95 (position paper).

MISSION
OF
ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.